The Design and Analysis of Algorithms	Take-home midterm
CS 681 Fall 2003	Due Wednesday, October 22, 2003

The midterm is individual work. Please do not discuss these questions with anyone expect with Swamy, Tom and Eva. We will do our best to answer your emails promptly during the week, and also will have extra office hours (will be posted on the Web).

(1) Consider an assignment problem, where we have a set of n stations that can provide service, and there is a set of some k request for service. One example is when the stations are cell-towers, and requests are cell-phones. Each request can be served by a given set of stations. The problem so far can be represented by a bipartite graph G: one side is the stations, the other the customers, and there is an edge (x, y) between customer x and station y if customer x can be served from station y. Assume that each station can serve at most one customers. Using a max-flow computation we can decide whether or not all customers can be served (we used this on Problem Set 3). In fact, using the same max-flow graph one can get an assignment of a subset of customers to stations maximizing the number of served customers.

However, in this problem, there is an additional complication: each customer offers a different amount of money for the service. Let U be the set of customers, and assume that customer $x \in U$ is willing to pay $v_x \geq 0$ for being served. Now the goal is to find a subset $X \subset U$ maximizing $\sum_{x \in X} v_x$ such that there is an assignment of the customers in X to servers.

Someone is recommending the following greedy approach. Consider customers in order of decreasing value (breaking ties arbitrarily). When considering customer x the algorithm will either "promise" service to x or reject x in a greedy fashion: Let X be the set of customers that so far have been promised service. We add x to the set X if and only if there is a way to assign $X \cup \{x\}$ to servers. Note that rejected customers will not be considered later. (This is viewed as an advantage: if we need to reject a high paying customers, at least we can tell him/her early.) However, we do not assign accepted customers to servers in a greedy fashion: we only fix the assignment after the set of accepted customers is fixed.

(a.) Does this greedy approach produce an optimal set of customers? Prove, or provide a counter example.

(b.) Would the analogous greedy approach also work for pairing up customers? In this version of the problem, we are again given a bipartite graph (say man and women on the two sides). Here each node x has value $v_x \geq 0$ (not only nodes on one side), and the goal is to find a matching M that maximizes $\sum_{x \in M} v_x$, where we use $x \in M$ to denote that x is matched by M. The greedy approach would again consider nodes in decreasing order of their costs, and accept or reject them greedily. Let X be the set of accepted nodes so far (now X has members on both sides of the bipartite graph), and we accept a new node x if there is a matching M where all nodes in $X \cup \{x\}$ are matched. Notice that M may also match some node outside of $X \cup \{x\}$). For example, the first node is accepted if it has an adjacent edge (independent of the value of the other end of the edge).

(2) Assume we are given n intervals, where interval i is $[s_i, f_i]$ (representing, say, the interval of time from s_i to f_i). The problem is to set up a data structure that allows you to answer a few queries efficiently. Here is the list of queries:

- *search*(s, t) should return "yes" if there is an $[s, t]$ interval in the set and "no" otherwise.
- *delete*(s, t) deletes the interval $[s, t]$ from the set assuming that it was in.
- *add*(s, t) adds the interval $[s, t]$ to the set, assuming it was not in, though intervals in the set may have $s_i = s$ or $t_i = t$, but not both.
- *start*(x) should return the number of intervals $[s_i, t_i]$ that have start time at most x (that is, $s_i \leq x$).
- *end*(x) should return the number of intervals $[s_i, t_i]$ that have end time at most x (that is, $x \leq t_i$).
- *number*(x) should return the number of intervals that contain the point x (an interval $[s_i, t_i]$ contains x if $s_i \leq x \leq t_i$).

Design a data structure that can do a sequence of m of these operations in $O(m \log n)$ time where n is the number of intervals involved in the data structure (intervals that were in the data structure at any time).

(3) In a lot of numerical computations, we can ask about the "stability" or "robustness" of the answer. This kind of question can be asked for combinatorial problems as well; here's one way of phrasing the question for the minimum spanning tree problem.

Suppose you are given a graph $G = (V, E)$, with a cost c_e on each edge e. We view the costs as quantities that have been measured experimentally, subject to possible errors in measurement. Thus, the minimum spanning tree one computes for G may not in fact be the "real" minimum spanning tree.

Given error parameters $\varepsilon > 0$ and $k > 0$, and a specific edge $e' = (u, v)$ that is not in the minimum spanning tree, you would like to be able to make a claim of the following form:

> $(*)$ Even if the cost of *each* edge were to be changed by at most ε (either increased or decreased), and the costs of k of the edges *other than* e' were further changed to arbitrarily different values, the edge e' would still not belong to any minimum spanning tree of G.

Such a property provides a type of guarantee that e' is not likely to be belong to the minimum spanning tree, even assuming significant measurement error (unless the cost $c_{e'}$ of the edge e' contains significant error).

(a.) Give a polynomial-time algorithm that takes G, e', and ε, and decides whether or not property $(*)$ holds for e' with $k = 0$.

(b.) Give a polynomial-time algorithm that takes G, e', ε, and k, and decides whether or not property $(*)$ holds for e'.